praxes Documentation

Release 0.7.1+5.gd5b91c6

Darren Dale

December 17, 2015

Contents

CHAPTER 1

Users' Guide

1.1 Introduction

This is the user guide for praxes.

1.2 Why Python?

Matlab claims to be "The Language of Technical Computing". It really is a nice collection of software. I learned how to program using Matlab, in order to settle a controversy relating to my graduate research project. As it turns out, I no longer use Matlab. There are two reasons: 1) Matlab is expensive, and once you begin developing tools using Matlab, you can quickly become committed to a big, long-term investment and 2) Python is much more pleasant to work with. The expressiveness, flexibility, and elegance are, in the opinion of many programmers, the most compelling reasons to use Python.

Scientific computing is currently undergoing something of a Renaissance in the Python community. It is now possible to create a compelling alternative to Matlab using NumPy, SciPy, IPython, Matplotlib, and a GUI toolkit like PyQt4. There are many really high-quality python libraries available, for example, h5py manages extremely large datasets by interfacing with the standardized hdf5 libraries, SymPy provides symbolic manipulation, VTK provides 3D visualization, and PyMol is a popular molecular visualization system which is frequently used to render the atomic structure of macro-molecules like proteins.

I think it is important to use and contribute to open source software, especially in the pursuit of science funded by the public. This speaks more broadly to the current state of scientific publishing, where the public pays for grants for scientific research, and then pay again to get access to the results of that research in an academic publication. I think it is important to maintain a degree of independence, and to have a sense of familiarity with the tools we depend upon. And finally, it has been my experience that open-source software development is an extremely efficient and successful, and results in the highest quality code. Eric Raymond makes a good case for the open-source development model in his book "The Cathedral and the Bazaar".

It takes much less time to develop and debug a program in Python than it would in a compiled language. It is true that some routines in Python are slow compared to compiled languages like C, but it is also true that once you understand a few Python idioms you can avoid many performance bottlenecks. Some cases still arise where you really need to squeeze every ounce of performance out of your hardware, and in that case you can implement a specific algorithm in a compiled in something like C and then wrap the resulting library for use from python. So in response to the question "Why Python?", my initial reaction is "why would anyone want to use anything else?".

1.3 Installation

Praxes depends on a few python packages, most of which can be installed on Linux using the distribution's package manager, or on Mac OS X with MacPorts. Note to windows users: you may need to run .exe installers as administrator by right-clicking on them and choose "run as Administrator"). Check that you have the following installed:

1. Python (version 2.6.x or 2.7.x. May already be installed on Linux)
2. Cython (version 0.15 or later, only required for Mac OS X and Linux)
3. NumPy (version 1.5.1 or later) [1]
4. PyQt4 (version 4.5.2 or later) [2]
5. PyMca (version 4.4.0 or later) [3]
6. matplotlib (1.1.0 or later) [1]
7. h5py (2.1.0 or later) [1], [4]
8. quantities (optional, only required by physref package)
9. Praxes

To install Praxes on OS X or Linux, download the source tar.gz file, unpack it, and run the following in the source directory:

```
python setup.py build && sudo python setup.py install
```

[1] Windows installers for 64-bit Python environments can be found here

[2] May require installing Qt on Mac, and development tools like pyqt4-dev and pyqt4-dev-tools through the package manager on Linux.

[3] Mac and linux users please install from source: e.g. pymca4.4.1-src.tgz. Windows users should follow the *PythonPackages* link, and download the file that includes the platform and python version in the name: e.g. PyMca-4.4.1.win-amd64-py2.7.exe.

[4] May require installing hdf5 on Linux and OS X, and development libraries like libhdf5-dev if installing with a packager manager on some linux distributions.

CHAPTER 2

Praxes API

2.1 io

2.1.1 Introduction

2.1.2 spec — Core tools for reading spec data files

The spec module provides an interface to data stored in files created by Certified Scientific's "spec" program.

Files are opened using the *open()* function, which returns a read-only dictionary-like interface to the scans contained in the file:

```
>>> from praxes.io import spec

>>> f = spec.open('spec_file.dat')
```

Each scan is also a read-only dictionary-like interface to the scalar and vector data.

At the top of the spec hierarchy is the *Mapping* class, which provides a dictionary interface similar to the dictionaries in python-3. Extending *Mapping* is *SpecFile*, which scans the file and creates an index of available scans:

```
>>> f.keys()
dict_keys(['1'])

>>> scan = f['1']
```

SpecFile.update() is provided to update the file's index in the event that data has been appended to the file.

Also extending *Mapping* is *SpecScan*, which scans a portion of the file and creates an index of available datasets and metadata:

```
>>> scan.keys()
dict_keys(['motor1', 'Epoch', 'Seconds', 'counter'])
```

Ordinary dictionary access of *SpecScan* yields proxies to the underlying data, which can be indexed to yield in-memory copies of the data:

```
>>> counter = scan['counter'] # counter is a proxy, no data has been loaded
>>> counter[...]
array([100, 101, 102])
>>> counter[0]
100
```

SpecScan.data provides another means of accessing the scalar data:

```
>>> scan.data[:, 0] # return the first column of data
>>> scan.data[3, :] # return the fourth row of data
```

Note that vector data (keys starting with "@") is not accessible using this mechanism.

If data has been appended to the file, the existing proxies will reflect this change:

```
>>> f.update() # or scan.update()
>>> counter[...]
array([100, 101, 102, 103])
```

Note, however, that the indices for the file and the scans are not completely reconstructed. They are only updated based on the assumption that data has only been appended to the file, and that any existing data in the file has not been modified.

SpecScan stores scan metadata in a read-only dictionary, which can be accessed using the *SpecScan.attrs* attribute:

```
>>> scan.attrs.keys()
dict_keys(['command', 'date'])
>>> scan.attrs['command']
'dscan motor1 -1 1 10 1'
```

Module Interface

praxes.io.spec.**open**(*file_name*)
Open *file_name* and return a read-only dictionary-like interface. If the file cannot be opened, an `IOError` is raised.

class praxes.io.spec.**Mapping**
The base class for all `spec` dictionary-like access to read-only data.

len(d)
Return the number of items in the dictionary *d*

d[key]
Return the item of *d* with key *key*. Raises a `KeyError` if *key* is not in *d*.

key in d
return `True` if *d* has a key *key*, else `False`.

get(*key*[, *default=None*])
Return the value for *key*, or return *default*

keys()
Return a new view of the keys.

items()
Return a new view of the `(key, value)` pairs.

values()
Return a new view of the values.

class praxes.io.spec.**SpecFile**
A class providing high-level access to scans stored in a "spec" data file. It inherits *Mapping*.

update()
Updates the file's index of scans in the file, if necessary. Also updates the indices for the scans in the file.

class praxes.io.spec.**SpecScan**
A class providing high-level access to datasets associated with a scan in a "spec" data file. It inherits *Mapping*.

attrs
A *Mapping* instance containing the metadata for the scan.

data
A proxy providing access to the scan's scalar data.

update()
Updates the scan's index of the data in the file, if necessary.

2.1.3 phynx — Object-oriented interface to hdf5 data files

Introduction

Phynx is a high-level interface to HDF5 files, built around the h5py bindings to the the HDF5 library. HDF5 is an cross-platform, open-source, binary file format designed for scientific data, allowing data to be organized in an hierarchy similar to the directories and files on a filesystem. It is fair to think of HDF5 as a file system for large, complex data. HDF5 provides access to data without loading the entire file in memory, thus it is easy to work with multi-gigabyte files on a computer with limited RAM.

Phynx offers a useful and intuitive object oriented interface designed to make it easier to work with the kind of data generated at synchrotron labs. The files are organized into a simple hierarchy that attempts to provide sufficient context for interpretation and processing.

The phynx file organization attempts to provide compatibility with the NeXus format. NeXus strives to provide a common exchange format for data generated at synchrotron and neutron facilities, using the NeXus API to interact with data stored in either XML or HDF5 files. NeXus application definitions play a major role in the "universality" of a NeXus file.

Many experiments, however, do not involve a well-defined application. They may involve combinations of application definitions, or instrumentation may need to be adapted to experimental necessity in ways that diverge from a well-defined application definition. Phynx is not necessarily suitable as a common exchange format. Rather, its design is based on a flexible, time-tested way to organize data which simplifies real-time analysis routines and allows data to be interpret in some other context not envisioned by a well-defined application definition. Data in phynx files can easily be exported into a more domain-specific common exchange format.

Exporting data

Naturally, you will feel more comfortable storing your data in the HDF5 format if you know how to export it to another familiar format. Imagine you have a file with the following standard format:

```
mydata.h5
  /entry_1/measurement/scalar_data/motor_1
                                  motor_2
                                  ion_chamber_1
                       /MCA/counts
                            deadtime
```

and you want to export channels 500-550 of the first three MCA spectra to an ASCII text file. You can do so with the following simple python code, which can be run as a script or in an interactive session:

```
from praxes.io.phynx import File
import numpy
```

```
f = File('mydata.h5')
mca_spectra = f['/entry_1/measurement/MCA/counts'][:3, 500:551]
numpy.savetxt('mca_spectra.txt', mca_spectra, fmt='%g')
```

Since we didnt need any of phynx special features, I could have done *from h5py import File* instead.

If you don't know how to run a python script, try reading the tutorial at the python website. For working with data interactively, have a look at the IPython package. If you don't know what *fmt='%g'* meant, you can find an explanation in the python documentation for string formatting.

2.2 praxes.combi

2.3 praxes.physref

2.3.1 praxes.physref.elam

The `elam` module provides an interface to the Elam x-ray database. Elements are accessed using `atomic_data`, which provides a mapping to the element data:

```
>>> from praxes.physref import elam
>>> copper = elam.atomic_data['Cu']
>>> print(copper.atomic_number)
29
```

Each element provides a mapping to the x-ray states reported in the Elam database:

```
>>> print(copper.keys())
['K', 'L1', 'L2', 'L3', 'M1', 'M2', 'M3', 'M4', 'M5']
>>> print(copper['K'].fluorescence_yield)
0.441091
```

Each x-ray state provides a mapping to the transitions originating from that state:

```
>>> print(copper['K'].keys())
['L1', 'L2', 'L3', 'M2', 'M3', 'M4,5']
>>> print(copper['K']['L3'].iupac_symbol)
'K-L3'
```

There is also a set of top-level functions in the `elam` module for calculating some simple properties of compositions, including conversions between stoichiometry and mass fractions, photoabsorption cross section, transmission and attenuation.

Note, in multithreading environments, there are issues sharing sqlite data between threads. As a result, objects arising from a given instance of `AtomicData`, such as `atomic_data`, should not be shared between threads. Instead, you should create a new instance of `AtomicData` in each thread to access the data.

Module Interface

Composition Functions

2.3.2 praxes.physref.waasmaier

The waasmaier module provides an interface to the energy-independent atomic form factors, as calculated by D. Waasmaier and A. Kirfel in Acta. Cryst. vA51, p416 (1995). The calculations take the form

$$f_0(s) = \sum_{i=0}^{5} a_i e^{-b_i s^2} + c$$

This approximation is valid for $s6^{-1}$, or $|Q|75^{-1}$.

Atomic form factors accessed using atomic_data:

```
>>> from praxes.physref.waasmaier import atomic_data
>>> import quantities as pq
>>> f0 = atomic_data['O']
>>> print f0(0 * 1/pq.angstrom)
7.99706
>>> print f0([0,1,2] * 1/pq.angstrom)
array([ 7.999706  ,  7.50602869,  6.31826029])
>>> f0 = atomic_data['O2-']
>>> print f0(0* 1/pq.angstrom)
9.998401
```

Module Interface

class praxes.physref.waasmaier.atomicdata.**AtomicData**

get(*item*, *default=None*)
Return the value for *key*, or return *default*

items()
Return a new view of the (key, value) pairs

iteritems() → an iterator over the (key, value) items of D

iterkeys() → an iterator over the keys of D

itervalues() → an iterator over the values of D

keys()
return a new view of the keys

values()
return a new view of the values

class praxes.physref.waasmaier.atomicdata.**FormFactor**(*symbol*, *db*)
FormFactor instances are callable to calculate the energy-independent form factors. They should be called with a Quantity in units of 1/length.

2.4 praxes.instrumentation

2.4.1 Introduction

2.5 praxes.config

2.5.1 Introduction

CHAPTER 3

Developer's Guide

3.1 Common Exchange Format

Some interested parties have been discussing the development of a common exchange format. What follows is a discussion of the format I developed for use at CHESS, and some of the considerations that lead to its current design.

3.1.1 Data Organization

How to organize the various data that are collected at synchrotron beamlines?

There seems to be general agreement that hdf5 is a good foundation on which to build a common data format for synchrotron experiments.

One proposal has been to pack all the motor positions, counters, etc., into a single array. If there are 10 such datasets and a scan is 100 points long, such an array would have 100 rows and 10 columns. I do not favor this approach, because it is difficult to convey context. How do we communicate what names are associated with each column? What about quantum efficiencies for counters, units for positioners? What about non-scalar data, like images, and multichannel analyzers? And what about the scalar data associated with such multidimensional datasets, like dead time?

For this reason, I prefer to make more extensive use of hdf5 groups and datasets. For simple datasets, like a positioner or a counter, a simple hdf5 dataset can be used, with additional metadata communicated by the hdf5 attributes associated with the dataset. For example, using h5py:

```
my_group['motor1'] = [0, .5, 1]
my_group['motor1'].attrs['units'] = 'mm'

my_group['monitor'] = [1, 1, 1]
my_group['monitor'].attrs['efficiency'] = 1e-5
```

For complex datasets, like an energy-dispersive detector, several datasets usually need to be grouped together in order to provide all the information required to interpret the data:

```
mca_group = my_group.create_dataset('vortex')
mca_group['counts'] = numpy.zeros(3, 1024)
mca_group['dead_time'] = [0, 0, 0]
mca_group['dead_time'].attrs['format'] = 'fraction'
mca_group['bins'] = numpy.arange(1024)
```

3.1.2 Data Structure

How should the datasets be shaped? Some scans are linear, some are regularly-gridded area scans, some may be more complex, like spiral tomography. What all scans have in common, however, is that data are acquired as a stream of points, one after another as time progresses.

For this reason, I have opted for the outer dimension of all datasets to have the same length as the number of points in the scan. Scalar data from a linear scan with 100 points will have shape (100,). Vector data, like MCA counts with 1024 bins, would have shape (100, 1024). Both of these datasets would have the exact same shape for an area scan with 10 points in each direction.

An additional benefit is that it simplifies data processing. The same routines can be used to iterate over scans of any shape.

How the data is interpreted or visualized is up to the application. Simple area scans can simply reshape the array. For the case of spiral CT, the application will obviously have to do some work in order to visualize the data, but what alternative is there for how to store the data in the file?

3.1.3 Context

In order for applications to interpret, analyze, and visualize the data, some additional context needs to be provided. How to differentiate a 100-point linear scan from a 10-by-10 area scan? The top-level group for the scan can contain an attribute like 'acquisition_shape' that communicates this context.

At CHESS, we still use Spec for data acquisition, which allows arbitrarily-named motors, counters, etc. Scans can be arbitrarily-structured as well, in the sense that any motor can be scanned. Scans often require compound motions. The way I have provided this information is to provide two attributes: "axis" and "primary". The "axis" attribute specifies the order of the axes: a value of 1 indicates it is the fastest-moving direction, 2 indicates it is the second-fastest-moving direction. The "primary" attribute communicates the ordering of axes in a scan involving compound motions.

Some datasets contain attributes that are specific to axes (or positioners), and others contain attributes that are specific to signals. Groups and datasets therefore have an attribute to communicate what kind of data they are.

This

3.2 Documenting Praxes

3.2.1 Getting started

Praxes' documentation is generated from ReStructured Text, using the Sphinx documentation generation tool. Sphinx-1.0.3 or later is required.

The documentation sources are found in the `doc/` directory in the trunk. The output produced by Sphinx can be configured by editing the `conf.py` file located in the `doc/` directory. To build the users guide in html format, run from the doc directory:

```
make html
```

and the html will be produced in `doc/_build/html/`.

3.2.2 ReSructuredText markup

The Sphinx website contains plenty of documentation concerning ReST markup and working with Sphinx in general. Here are a few additional things to keep in mind:

- Please familiarize yourself with the Sphinx directives for inline markup. Praxes' documentation makes heavy use of cross-referencing and other semantic markup. For example, when referring to external files, use the `:file:` directive.
- Function arguments and keywords should be referred to using the *emphasis* role. This will keep Praxes' documentation consistant with Python's documentation:

```
Here is a description of *argument*
```

Please do not use the *default role*:

```
Please do not describe `argument` like this.
```

nor the `literal` role:

```
Please do not describe ``argument`` like this.
```

- Sphinx does not support tables with column- or row-spanning cells for latex output. Such tables can not be used when documenting Praxes.
- Mathematical expressions can be rendered as png images in html, and in the usual way by latex. For example:

`:math:'\sin(x_n^2)'` yields: $\sin(x_n^2)$, and:

```
.. math::

  \int_{-\infty}^{\infty}\frac{e^{i\phi}}{1+x^2\frac{e^{i\phi}}{1+x^2}}
```

yields:

$$\int_{-\infty}^{\infty} \frac{e^{i\phi}}{1+x^2\frac{e^{i\phi}}{1+x^2}}$$

- Footnotes [1] can be added using `[#]_`, followed later by:

```
.. rubric:: Footnotes

.. [#]
```

- Use the *note* and *warning* directives, sparingly, to draw attention to important comments:

```
.. note::
   Here is a note
```

yields:

Note: here is a note

also:

Warning: here is a warning

- Use the *deprecated* directive when appropriate:

```
.. deprecated:: 0.98
   This feature is obsolete, use something else.
```

yields:

Deprecated since version 0.98: This feature is obsolete, use something else.

[1] For example.

- Use the *versionadded* and *versionchanged* directives, which have similar syntax to the *deprecated* role:

```
.. versionadded:: 0.98
   The transforms have been completely revamped.
```

  New in version 0.98: The transforms have been completely revamped.

- Use the *seealso* directive, for example:

```
.. seealso::

   A bit about :ref:`referring-to-praxes-docs`:
      One example
```

  yields:

  See also:

  A bit about *Referring to Praxes documents*: One example

- Please keep the *Glossary* in mind when writing documentation. You can create a references to a term in the glossary with the `:term:` role.
- The autodoc extension will handle index entries for the API, but additional entries in the index need to be explicitly added.

3.2.3 Docstrings

In addition to the aforementioned formatting suggestions:

- Please limit the text width of docstrings to 70 characters.
- Keyword arguments should be described using a definition list.

Note: Praxes makes extensive use of keyword arguments as pass-through arguments, there are a many cases where a table is used in place of a definition list for autogenerated sections of docstrings.

3.2.4 Figures

Dynamically generated figures

The top level `doc` dir has a folder called `pyplots` in which you should include any pyplot plotting scripts that you want to generate figures for the documentation. It is not necessary to explicitly save the figure in the script, this will be done automatically at build time to insure that the code that is included runs and produces the advertised figure. Several figures will be saved with the same basnename as the filename when the documentation is generated (low and high res PNGs, a PDF). Praxes includes a Sphinx extension (`sphinxext/plot_directive.py`) for generating the images from the python script and including either a png copy for html or a pdf for latex:

```
.. plot:: pyplot_simple.py
   :include-source:
```

The `:scale:` directive rescales the image to some percentage of the original size, though we don't recommend using this in most cases since it is probably better to choose the correct figure size and dpi in mpl and let it handle the scaling. `:include-source:` will present the contents of the file, marked up as source code.

Static figures

Any figures that rely on optional system configurations need to be handled a little differently. These figures are not to be generated during the documentation build, in order to keep the prerequisites to the documentation effort as low as possible. Please run the `doc/pyplots/make.py` script when adding such figures, and commit the script **and** the images to svn. Please also add a line to the README in doc/pyplots for any additional requirements necessary to generate a new figure. Once these steps have been taken, these figures can be included in the usual way:

```
.. plot:: tex_unicode_demo.py
   :include-source
```

3.2.5 Referring to Praxes documents

In the documentation, you may want to include a document from the Praxes source, like a license file or an example. When you include these files, include them using the `literalinclude` directive:

```
.. literalinclude:: ../examples/some_example.py
```

3.2.6 Internal section references

To maximize internal consistency in section labeling and references, use hyphen-separated, descriptive labels for section references, eg:

```
.. _howto-webapp:
```

and refer to it using the standard reference syntax:

```
See :ref:`howto-webapp`
```

Keep in mind that we may want to reorganize the contents later, so let's avoid top level names in references like `user` or `devel` or `faq` unless necesssary, because for example the FAQ "what is a backend?" could later become part of the users guide, so the label:

```
.. _what-is-a-backend
```

is better than:

```
.. _faq-backend
```

In addition, since underscores are widely used by Sphinx itself, let's prefer hyphens to separate words.

3.2.7 Section names, etc

For everything but top level chapters, please use `Upper lower` for section titles, eg `Possible hangups` rather than `Possible Hangups`

3.3 Coding style guide

Read PEP 8

3.4 Releases

Before creating a release, the version number needs to be updated in `praxes/__init__.py`. Ensure that the Praxes source directory appears on the $PYTHONPATH, so the package version numbers will be advertised correctly for the installers and the documentation.

3.4.1 Creating Source Releases

Praxes is distributed as a source release for Linux and OS-X. To create a source release, just do:

```
git tag v{X}
git push --tags
```

This automatically creates links to download the source release at the Praxes downloads page.

3.4.2 Creating Windows Installers

Open a DOS window, cd into the praxes source directory and run:

```
python setup.py bdist_wininst --install-script=praxes_win_post_install.py
```

This creates the windows installer in the *dist/* directory, which can then be uploaded to the Praxes downloads page.

3.4.3 Publishing Praxes' documentation

When publishing a new release, the Praxes doumentation needs to be generated and published as well. Sphinx is required to build the documentation. First, run:

```
git clean -fdx
```

Then, in the `doc/` directory, run:

```
make html
```

Next, move to the master branch of the praxes.github.com repository, and run:

```
cp -r ../praxes/doc/_build/html/* .
git status
```

Use `git add` to add any new files to the repository, and then commit and push the changes to the upstream praxes repository:

```
git commit -a -m "meaningful commit message"
git push
```

and visit the Praxes documentation page to view the documentation.

CHAPTER 4

About Praxes

4.1 Credits

Praxes was started and continues to be led by Darren Dale.

4.1.1 Core developers

As of this writing, core development team consists of the following developers:

Darren Dale <darren.dale-AT-cornell.edu> Project creator and leader, Praxes core, testing, documentation, release manager.

4.1.2 Contributors

And last but not least, all the kind Praxes contributors who have contributed new code, bug reports, fixes, comments and ideas. A brief list follows, please let us know if we have omitted your name by accident:

Jeffrey Lipton Preliminary development of graphical user interfaces for plotting and moving motors via specclient.

4.1.3 Special thanks

The Praxes project is also very grateful to:

Guido van Rossum and the Python development team For an incredible programming language.

The Scientific Python community especially the developers of NumPy, SciPy, Matplotlib, IPython, h5py, and the developers at Enthought (http://www.enthought.com).

Phil Thompson and Riverbank Computing For PyQt4 and dip.

4.2 History

4.2.1 Origins

Praxes was started in 2007 (under another name) by Darren Dale while he was a staff scientist at the Cornell High Energy Synchrotron Source. Darren began using Python as an open alternative to Matlab, and started Praxes to provide a flexible environment for integrating the various tools required for analysis of data collected at synchrotron laboratories.

4.3 License and Copyright

Praxes is licensed under the terms of the new or revised BSD license, as follows.

4.3.1 License

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- Neither the name of the Praxes Development Team nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

4.3.2 About the Praxes Development Team

Darren Dale began Praxes in 2007 and is the project lead.

The Praxes Development Team includes all contributors to the Praxes project. Currently active contributors include:

- Darren Dale (project leader)

4.3.3 Copyright Policy

Praxes uses a shared copyright model. Each contributor maintains copyright over their contributions to Praxes. These contributions are typically changes (diffs/commits) to the repositories. The source code in its entirety is not the copyright of any single person or institution. Instead, it is the collective copyright of the entire Praxes Development Team. Individual contributors wishing to maintain a record of contributions to which they hold specific copyright should indicate their copyright in the commit message of the change to the repository.

Any new code contributed to Praxes must be licensed under the BSD license or a similar (MIT) open source license.

4.3.4 Miscellaneous

Praxes is built upon PyQt4 and dip, which are provided by Riverbank Computing (http://www.riverbankcomputing.com/news) under the terms of the GNU Public License. Riverbank Computing explicitly grants additional rights to license such derived work under the terms of the BSD license (among others). This allows Praxes to be developed and distributed under the terms of the BSD license. Commercial projects can be derived from Praxes, but if such a project continues to derive from PyQt4, a commercial license is required from Riverbank Computing.

CHAPTER 5

Glossary

AGG The Anti-Grain Geometry (Agg) rendering engine, capable of rendering high-quality images

EPS Encapsulated Postscript (EPS)

JPG The Joint Photographic Experts Group (JPEG) compression method and file format for photographic images

numpy numpy is the standard numerical array library for python, the successor to Numeric and numarray. numpy provides fast operations for homogeneous data sets and common mathematical operations like correlations, standard deviation, fourier transforms, and convolutions.

PDF Adobe's Portable Document Format (PDF)

PNG Portable Network Graphics (PNG), a raster graphics format that employs lossless data compression which is more suitable for line art than the lossy jpg format. Unlike the gif format, png is not encumbered by requirements for a patent license.

PS Postscript (PS) is a vector graphics ASCII text language widely used in printers and publishing. Postscript was developerd by adobe systems and is starting to show its age: for example is does not have an alpha channel. PDF was designed in part as a next-generation document format to replace postscript

pyqt pyqt provides python wrappers for the *Qt* and *Qt4* widgets library. Widely used on linux and windows; many linux distributions package this as 'python-qt3' or 'python-qt4'.

python python is an object oriented interpreted language widely used for scripting, application development, web application servers, scientific computing and more.

Qt Qt is a cross-platform application framework for desktop and embedded development.

Qt4 Qt4 is the most recent version of Qt cross-platform application framework for desktop and embedded development.

raster graphics Raster graphics, or bitmaps, represent an image as an array of pixels which is resolution dependent. Raster graphics are generally most practical for photo-realistic images, but do not scale easily without loss of quality.

SVG The Scalable Vector Graphics format (SVG). An XML based vector graphics format supported by many web browsers.

TIFF Tagged Image File Format (TIFF) is a file format for storing images, including photographs and line art.

vector graphics vector graphics use geometrical primitives based upon mathematical equations to represent images in computer graphics. Primitives can include points, lines, curves, and shapes or polygons. Vector graphics are scalable, which means that they can be resized without suffering from issues related to inherent resolution like are seen in raster graphics. Vector graphics are generally most practical for typesetting and graphic design applications.

p

V